NASA Technical Memorandum 106238

Portable Programming on Parallel/Networked Computers Using the Application Portable Parallel Library (APPL)

Angela Quealy
Sverdrup Technology, Inc.
Lewis Research Center Group
Brook Park, Ohio

and

Gary L. Cole and Richard A. Blech
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio

July 1993

NASA

PORTABLE PROGRAMMING ON PARALLEL/NETWORKED COMPUTERS USING THE APPLICATIONS PORTABLE PARALLEL LIBRARY (APPL)

Angela Quealy
Sverdrup Technology, Inc.
Lewis Research Center Group
Brook Park, Ohio 44142

and

Gary L. Cole and Richard A. Blech
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

ABSTRACT

The Application Portable Parallel Library (APPL) is a subroutine-based library of communication primitives that is callable from applications written in FORTRAN or C. APPL provides a consistent programmer interface to a variety of distributed and shared-memory multiprocessor MIMD machines. The objective of APPL is to minimize the effort required to move parallel applications from one machine to another, or to a network of homogeneous machines. APPL encompasses many of the message-passing primitives that are currently available on commercial multiprocessor systems. This paper describes APPL (version 2.3.1) and its usage, reports the status of the APPL project, and indicates possible directions for the future. Several applications using APPL are discussed, as well as performance and overhead results.

INTRODUCTION

Computer simulations of multidisciplinary, grand challenge problems in aerospace propulsion involve considerable programming effort and computing time. Reductions in simulation time and cost can be achieved by computing portions of the simulation in parallel, using multiprocessor systems. Many different multiprocessor systems are available, each having a unique programming environment that takes a significant amount of time to learn. In the future, it is possible that a distributed computing environment will be used, which will complicate the programming effort even more. The Application Portable Parallel Library (APPL) project was initiated to facilitate the exploitation of parallel computer technology, by minimizing the learning curve and programming effort required to move application codes to different and/or networked machines. It was also intended to provide a basis for building other software tools and utilities to support parallel processing.

A distributed memory message-passing model has been chosen as the basis for APPL, because it can be used on the variety of machines on the market today, including distributed memory, shared memory, and hybrid (cluster model) machines. APPL allows the user to develop application programs written only in terms of communicating processes, and thus design programs independent of target hardware. Basic message-passing primitives comprising APPL are used to communicate between processes, and the primitives are the same regardless of the machine. Care has been taken to minimize the overhead incurred with using portable, rather than the native, primitives of the particular target. As a result, the user is free to map an application to any particular hardware configuration (from stand-alone machines to a network of homogeneous machines) merely by specifying a process definition file. A special initiator task takes care of initializing the environment and loading the executable code. This combination of portable message-passing primitives, a process definition file, and an initiator task yield an application that is portable. Tools built on top of APPL become portable as well.

This research was supported by NASA Lewis Research Center under contract NAS3-25266 with Gary L. Cole as monitor.

This report describes the development approach and the APPL version 2.3.1 software. An example problem is then developed to illustrate the features of APPL. Finally, the status of the project, as well as applications and performance, are discussed, with concluding remarks to indicate possible directions for the future of the project.

APPL DEVELOPMENT APPROACH

The initial effort in developing a portable capability at NASA Lewis Research Center involved examining a number of portable libraries that already existed in the public domain. Commercial packages were not seriously considered because of potential restrictions by commercial licensing agreements when building other tools on top of the portable library. The intent was to acquire the most suitable portable library and adapt it to the computing environment at NASA Lewis. The libraries examined were PICL from Oak Ridge National Laboratory (ref. 1), and PARMAC from Argonne National Laboratory (ref. 2). Early versions of the TCGMSG and P4 libraries from Argonne were also examined. TCGMSG and P4 were both developed there as outgrowths of PARMAC, and have evolved since the development of APPL (refs. 3, 4).

A review of the available libraries helped establish the features desired for APPL, including:

- A well-defined set of primitives
- Support for applications programmed in Fortran or C
- Synchronous and asynchronous communication
- Support on both shared and distributed memory machines
- Communication via shared memory rather than via sockets on shared memory machines

At the time, none of the existing libraries completely satisfied these requirements, but they did provide a good basis for what was needed. The TCGMSG package was chosen as the basis for APPL, with portions of the P4 package filling in for capabilities which TCGMSG lacked. The effort has focused on blending these libraries into a single library, by augmenting and/or modifying where required, to satisfy the above requirements. Modifications to the original Argonne packages have included performance and functionality enhancements, code re-structuring, and the addition of a more versatile user interface.

APPL SOFTWARE

Overview of Commands and Parameters

The number of communication primitives included in APPL have been kept to a minimum. The primitives included in APPL are those operations found to be essential to writing a parallel application. These primitives comprise a common subset of message-passing and support operations found on distributed memory machines today. Potential portability across intended target machines was a factor in selecting which primitives to include in APPL. Additional operations may be added in the future.

It is hoped that eventually a message-passing standard will emerge across all commercial platforms. Such an effort has begun (ref. 5), and it should be noted that the communication primitives included in APPL are a subset of the elements included in the discussions regarding a standard message-passing library. When a standard does emerge, applications developed using APPL should be able to adopt this standard without much effort.

The APPL low-level primitives are listed in table 1, along with a brief description. High-level primitives, built on top of the low-level primitives, are listed in table 2. Most of the primitives are self-descriptive; however, the functionality of the different send and receive communication primitives is discussed below.

The APPL *ssend* primitive is a fully blocking send operation. The sending process sends a message and then blocks until that message has been received by the receiving process. The APPL *srecv* primitive is a blocking receive operation. The receiving process blocks until the appropriate message, selected by a message type, arrives. Transparent to the user, the receiving process returns an acknowledgement of the receipt of the message to the sending process. Once acknowledged, the sending process is free to continue execution. A fully blocking send/receive operation is also referred to as synchronous communication. Caution must be applied here, in order to avoid deadlock situations.

The APPL *asend* routine is a partially blocking send operation. The sender returns from the *asend* operation once the message being sent has vacated the message buffer it had been occupying. This type of send operation is sometimes referred to as an asynchronous send operation. The APPL *arecv* routine is a blocking receive operation. The receiving process blocks until the appropriate message, selected by a message type, arrives. There is essentially no difference between the *srecv* and *arecv* primitives from a user's point of view, and in future releases of APPL, these primitives may be merged into a single *receive* primitive. The underlying difference between these primitives is that the *arecv* primitive does not return an acknowledgement of the message arrival to the sender.

At the moment, APPL does not support a non-blocking receive operation, such as the Intel iPSC/860 *irecv* operation, or a non-blocking send operation, such as the Intel iPSC/860 *isend* operation (ref. 6). These operations may be added in the future as required. Also, the various APPL send and receive operations are not intermixable. The *asend* primitive must be used in conjunction with the *arecv* primitive, and the *ssend* primitive must be used in conjunction with the *srecv* primitive; however, this feature may also change in a future release of APPL.

Process Definition File

The APPL model considers an application to be a collection of communicating processes. Thus the application program is written to match the structure of the problem, rather than the structure of the target architecture on which the application will be executed. A programmer should not have to be concerned with the underlying interconnection network of a potential target machine. If such machine idiosyncracies were included in programming an application, that code would require modification if ported to a machine with a different interconnection network. For instance, APPL contains no gray code mapping functions. If an application were programmed using a gray code mapping scheme, and then moved to another target, this mapping scheme would be meaningless, and would require recoding.

APPL attempts to eliminate such machine idiosyncracies and architecture-specific issues from the application code development phase, resulting in a code that is not only portable, but also "architecture-independent." However, given the variety of parallel architectures, including distributed memory, shared memory, and "cluster model" architectures, a method is required to map a portable application either to a particular target architecture, or across a network of machines. This method would describe how an application is distributed across a network of physical hardware. If the application were to be moved to a new machine, or a network of machines, only this description would change. The application code itself would not change.

In the APPL environment, once the portable application code has been written and is ready to be executed, a target machine is chosen, and a mapping between application and architecture is performed. A process definition file defines this mapping of application to architecture (see some examples of process definition files in figure 1). The executable images which compose the application are listed in the process definition file, along with the machine-specific requirements necessary to map each executable image to a physical processor. The information listed in a process definition file may include user name(s), host name(s), executable image name(s), the number of executable images in the application, the mapping style, and the working directory which contains the executable images and I/O files. The information required in the process definition file varies, depending on the selected target architecture.

Initiator Process

Once a process definition file has been defined that describes a particular application and its physical mapping, a method is required to start the application on the specific target machine. To accomplish this, APPL uses an initiator process, which reads the process definition file and starts the executable images on the appropriate physical pieces of hardware. The initiator process handles all machine-specific tasks required in starting and terminating an application, and further masks the implementation-specific details of executing an application on a particular architecture from the user.

To invoke the initiator process, the user simply types *compute*. The name of the process definition file may be supplied to *compute* using the '-p' option, or the default name 'procdef' can be used. Depending on the selected target architecture, the initiator process begins the machine-specific tasks required to access the required processors and loads the executable images of the application, as illustrated in figure 2. These tasks may vary greatly from machine to machine because of the diversity in APPL's target platforms. For instance, on the Intel iPSC/860, the initiator process performs the *getcube*, *load*, *waitall* and *relcube* operations. On a network of workstations, the initiator process does much more. In this situation, processes must be created using *fork/execv/rsh*, and message-passing data structures must be defined using calls to shared memory and semaphore manipulation routines. All of these machine-specific details are hidden from the user.

Implementation Details

The APPL source code consists of a set of library routines, written in C, which compose the APPL primitives, along with an initiator program, also written in C. Compiler directives surround machine-specific code, which allows the APPL code to be transported to a variety of machines, and simply recompiled to run on a particular machine. Thus the APPL source code itself is portable, in addition to all application codes developed using APPL.

The APPL machine implementations can be divided into three distinct subsets: distributed memory and shared memory implementations, and an implementation across a network of machines. The more important aspects of these three subsets are discussed below. Also, the relative ease of porting APPL to additional target platforms is discussed.

a. Distributed memory machine implementation

The distributed memory machine implementation is the simplest of the three subsets. Each APPL library call consists of a wrapper around a similar native message-passing call. Parameters are adjusted and checked for validity.

The initiator process for the distributed memory machines consists of a host program, which starts the execution of the application and waits for execution to terminate. On the Delta machine, a simple script is used in place of a host program. The initiator process reads the process definition file and determines the number of processors required for the application. The initiator then accesses the number of processes required and loads the executable images onto the specified processors. The initiator then idles, waiting for the application to complete. On completion, the initiator releases the processors and terminates.

b. Shared memory machine implementation

The shared memory machine implementation is more complex. Since there is no native message-passing library on these machines, one had to be implemented using the available features of each machine. System resources such as shared memory and semaphores are used to allow message-passing between multiple processes. Application processes are grouped into a cluster, with one application process in the cluster singled out as a cluster master, as illustrated in figure 3. Multiple clusters are permissible on a

single machine, however for performance considerations all processes are typically grouped into a single cluster.

The cluster master is created by the initiator process, using calls to *fork* and *execv*. The *pbegin* primitive is invoked in the application code, and works with the initiator to establish the proper environment to run the application. To establish this environment, the cluster master requires information about the application from the initiator. This information is generated by the initiator, or taken from the process definition file by the initiator, and includes the cluster communication port number, the executable image names of the remaining cluster processes, and the application working directory. The cluster master stores this information in a shared memory area, S1, which is accessible by all cluster processes.

A second, larger shared memory area, S2, is acquired and initialized by the cluster master. This area is also accessible by all cluster processes, and is used for the data structures required to enable message-passing. These data structures include sets of message buffers and message headers, a linked list indicating free message buffers and message headers, and a message queue for each process. The cluster master acquires a segment of shared memory large enough to hold one set of these data structures for each process in the cluster. Since these structures are in shared memory, message-passing between processes occurs when the sending process writes a message into the buffer and queue area of the destination process. A block/wakeup facility is implemented using a semaphore to enable waiting for messages without polling the receive queues.

Once all data structures are in place, the remaining cluster processes are created by the cluster master using *fork/execv*. Once all the application processes have started, the initiator idles until all the processes have completed. Then the initiator process terminates.

c. Implementation across a network of workstations

The APPL implementation on a network of workstations is similar to that of the shared memory machine implementation. When an application is run across a network of machines, there is typically one cluster of processes per machine. The data structures required for message-passing are created and initialized by the cluster master on each machine. However, rather than communicating between clusters via shared memory, information is transferred between Internet domain SOCK_STREAM sockets, which use TCP/IP over Ethernet. Multiple clusters may exist on any single workstation; however, communication between processes located on the same machine yet in different clusters will also occur via sockets. To achieve the best performance, all processes on a single machine should be grouped into a single cluster. Also, most TCP implementations try to 'batch-up' requests by waiting for some minimum amount of data to communicate. The NO_DELAY feature is used to avoid stream-oriented batching, and it improves performance considerably. The network proves to be the bottleneck in most applications which require substantial communication however, if a common Ethernet is used between workstations. This fact should be considered when implementing an application.

Each process cluster has a communication process whose function is to establish a communication link between clusters, and to receive messages from processes outside of the cluster. This communication process allows computational processes to continue independently of communication. If a connection request occurs, the communication process establishes a connection, and enters the appropriate information into a connection table.

Rather than each process establishing a connection to every other process, communicating processes are connected as required. Figure 4 illustrates a message transmission between process A and process D. It is assumed that A and D are in separate clusters, because if they were in the same cluster, communication would occur via shared memory. Process A must check its communication table to determine if a socket connection to process D's cluster already exists. If it does, then communication occurs across that path. If such a connection does not exist, process A initiates a connection to process D's cluster. This allows

process A to communicate with any other process in that cluster, so in this example, processes C and D are included. When a message arrives from a process outside of the cluster, the communication process, CP, receives the message and stores it in the appropriate message queue of the destination process, D. Process D can now receive this message as if the message had originated within its own cluster.

d. Future implementations

Because APPL has been implemented on these three classes of machines, porting APPL to new targets in these categories is a relatively simple procedure. On the distributed memory machines, the APPL primitives must be matched to the corresponding message-passing calls from the new target machine. On other machines, an APPL port which is similar to the desired port is chosen as a starting point. The implementor may have to resolve some compiler differences, and locate certain *include* files on a particular machine. It is assumed that a capability such as that provided by the UNIX System V semaphore and shared memory system calls (ref. 7) exists on most intended platforms. If such a capability does not exist, a socket-based message-passing implementation can be developed; however, this might require some re-structuring of APPL.

Tools

Figure 5 indicates that APPL is intended to be one piece of a larger programming environment. APPL was developed first, so that other tools and utilities could be built, and thus be portable by virtue of being built on top of APPL. Such tools are beginning to emerge (ref. 8).

In the absence of APPL-specific programming tools such as performance monitors and parallel debuggers, it has been found that other existing tools can be used in conjunction with APPL to round out the program development environment. For instance, Intel provides a Performance Analysis Tool (PAT) (ref. 9) for the iPSC/860 hypercube. A programmer can access PAT by linking the performance-monitoring library with an APPL program. Since APPL invokes the native message-passing library on the Intel iPSC/860, the calls to the performance-monitoring library routines are made as if the application had been written using the native library. The performance monitor results can then be interpreted for the APPL programs. Figure 6 illustrates a bar chart compiled using information obtained from using PAT on the ISTAGE code (ref. 10).

On the Silicon Graphics 4D/480 machine, the Power FORTRAN Accelerator (ref. 11) can be used to parallelize sections of code automatically. This accelerator concentrates on splitting FORTRAN DO loops across processors, and can be used in conjunction with APPL to achieve improved results on multiprocessor SGI machines. Also, SGI tools such as *gr_osview* (ref. 12) can assist a programmer in determining what is occurring during the execution of an application on a general level.

EXAMPLE PROBLEM

This section describes an example problem to illustrate the various features of APPL. The example, kept simple for explanation purposes, is a heat flow problem (ref. 13) which uses Laplace's equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

to determine the steady state temperature of a flat plate exposed to constant boundary conditions. The problem is stated as follows:

> A thin steel plate is 20 x 20 cm square. If one edge is held at 100° C, and the others are held at 0° C, what are the steady state temperatures at the interior points?

The flat plate has a certain initial temperature, and is subjected to a temperature at each of its four boundaries. To discretize this problem, the flat plate is divided into sections of size Δx by Δy. In this application, it is assumed that $\Delta x = \Delta y$. The resulting grid is now numbered so that each intersection represents a point of the flat plate at which a temperature can be evaluated. The boundary points are included in this numbering. A 14 x 14 grid is used for this example, and is illustrated in figure 7.

A central difference approximation is used to calculate the temperature $T_{i,j}$ at each interior grid point. All exterior grid points assume the boundary temperatures. The iterative equation

$$T_{i,j}^{(t+1)} = \frac{T_{i+1,j}^{(t)} + T_{i-1,j}^{(t)} + T_{i,j+1}^{(t)} + T_{i,j-1}^{(t)}}{4}$$

is used to calculate a temperature solution for each grid point (i,j). The value of $T_{i,j}$ at iteration (t+1) is an average of the temperatures of its four nearest neighbors at iteration (t). A solution is obtained when a specified convergence criterion is met.

Since the temperature value at each grid point is calculated from only the neighboring grid point values, the problem can be easily divided across multiple processors, as illustrated in figure 8. Communication is required to transmit boundary information between neighboring processes. Each process requires enough memory space to store its portion of the computational grid, along with two columns of boundary data.

A pseudo-code segment is included in figure 9 to illustrate how such a problem would be programmed using APPL primitives. There are two sections of interest. The first determines the neighboring processes of each process. The second is the main DO loop, which calculates each new temperature value for a specified number of iterations. On each iteration, the boundary temperature values are exchanged between neighbors, and the new temperature values are calculated by each process for their own portion of the computational grid. For simplicity in this example, a user-supplied maximum number of iterations is used, rather than testing for convergence.

Once this application is programmed using APPL primitives, the code is portable to the variety of architectures on which APPL is implemented. These architectures range from hypercubes, to meshes, to shared memory machines, to a network of workstations. At this point, an architecture is selected, and a process definition file is written, defining the mapping between architecture and application. Once a process definition file has been established, the initiator program, *compute*, is called, and the application is started on the selected target machine(s).

STATUS

APPL is currently implemented on the Intel iPSC/860, the Intel Delta machine, the Intel simulator, the Alliant FX/80, the Silicon Graphics IRIS4D series, the IBM RS6000 series, the Sun SparcStation series, and the Hypercluster (a NASA Lewis multi-architecture test bed (ref. 14)). Other target machines are being considered, and it is hoped that as more target implementations of APPL are developed outside of NASA Lewis, these implementations can be accessed and incorporated into APPL, to be distributed with the APPL software to other interested users.

APPL has also been implemented across a homogeneous network of workstations (SGI, IBM or Sun). Using this mode, each workstation can be used individually to run multiple processes, or workstations can be networked, with clusters of processes running on each workstation. The size of the cluster may vary from one machine to another. This capability allows great flexibility, and experience has shown it to be a very robust development environment. Once an application has been developed, it can be ported to a high-end machine such as the Intel iPSC/860, with no modification to the application code. This flexibility eases code development difficulties encountered with early releases of parallel machines.

Also, some researchers have limited access to state-of-the-art parallel systems, but have easier access to several UNIX-based workstations. A network of workstations may be the only option for studying parallel processing issues in these situations. Codes developed in this manner can then be easily ported to a parallel machine if one does become available in the future.

APPLICATIONS

APPL has been used to implement several applications, both at NASA Lewis and elsewhere. Three of these applications are now briefly described. Table 3 lists some of the other applications which have been developed using APPL.

Existing efforts at NASA Lewis involve developing parallel implementations of the average passage turbomachinery code (ref. 15). A parallel implementation of the inviscid version of the code, ISTAGE (ref. 10) has been developed. This simulation was configured for a single blade row, and a medium grain partitioning was used. The parallel ISTAGE code has been ported to a variety of machines, and the performance results are discussed below.

Currently two parallel implementations of MSTAGE, the viscous version of the average passage turbomachinery code, are being developed. This simulation is configured for multiple blade rows. A "blade row parallel" version of the code was developed initially, using a coarse grain partitioning to split the calculation of each blade row onto a separate processor. The performance results are discussed below for several platforms. Two medium grain partitionings are underway. One involves splitting the domain along the K (tangential) dimension, to achieve more parallelism. The other involves splitting the domain along the I (axial) dimension. It is hoped that these two versions can be merged in the future, for an even finer-grained partitioning of the application.

Existing efforts at Purdue University involve using APPL to develop a grid-oriented database for parallel processing of CFD applications (ref. 8). This database is a tool which helps to simplify the programming of data exchanges between grid blocks, taking into account the differences at grid boundaries. This tool is able to handle special situations such as a structured grid meeting an unstructured grid, or overlapping grids. The programmer is able to manipulate grids, rather than dealing with data on the lower, message-passing level. This database is an example of a tool which is portable by virtue of being developed on top of APPL.

Also at Purdue, an effort is underway to develop tools to assist in the load-balancing process. Using APPL across a network of IBM RS6000 workstations has proven to be an ideal environment for this task. Processes can be created and clustered on workstations with ease. On machines such as the Intel iPSC/860 hypercube, which allow only a single task per processor, applications must be written to handle the load-balancing issues. With APPL, the application can be written with the appropriate number of processes required by the application. The processes can then be clustered on workstations, depending on how each process contributes to the work load.

PERFORMANCE

The performance of several parallel implementations of applications using APPL are discussed here. Three performance issues are examined. First, the amount of overhead incurred when using APPL rather than the native message-passing library of a particular machine is studied. This overhead will determine the extent to which using APPL affects the performance of an application. Second, the communication rates between processes on all platforms, and on shared memory platforms in particular, is examined. The APPL implementation of message-passing via shared memory is contrasted with a socket-based implementation of message-passing. Finally, two application codes written using APPL were ported to a variety of architectures to determine the effect on performance incurred by porting the application from one

machine to another in such a generic manner. As a part of this study, the performance of applications running in parallel on shared memory machines and on networks of workstations using the APPL message-passing model is examined to determine if such a practice is feasible.

Overhead Measurements

Measuring the amount of overhead incurred by APPL on various parallel platforms involves executing the application written in the native message-passing library of the machine, and then executing the same application using APPL on that machine. Overhead is defined as the difference in wall clock time incurred using APPL rather than the native message-passing library, referenced to the wall clock time using the native message-passing library. Overhead was calculated on APPL's three distributed memory platforms (Intel iPSC/860, Delta, and Hypercluster), where a native message-passing library existed for comparison purposes. The application programs were written in FORTRAN. Table 4 lists the overhead incurred when using APPL rather than the native library for two test programs.

The first test program is a ring test, which passes a message of a specified size around a ring of 16 processors for 100 iterations. The ring test was expected to be the worst case scenario in terms of overhead, especially for smaller messages, since it is a completely communication-bound application. The overhead ranges from 23.6% for small message sizes for the worst case on the Delta, to 0.2% for large message sizes for the best case on the Intel iPSC/860. The overhead incurred on the Hypercluster is substantially less than that incurred on the commercial machines, which can be attributed to the slower processor speeds and link rates of the Hypercluster.

A more important concern was minimizing the overhead of an application typical of those used at NASA Lewis. The RVC code (ref. 16), a 2-D Euler code, was chosen as a typical application. The overhead incurred with running the RVC code using APPL falls within an acceptable range (0.6% to 3.5%) across all distributed memory platforms.

Communication Rate Measurements

The communication rate is defined as the number of bytes per second transmitted between two processes. This value is dependent on the speed of the processors, the speed of the link between processes, and the efficiency of the software used to transmit information between processes. Figure 10 illustrates the communication rates calculated using an APPL ring test on several platforms, for small, medium and large-sized messages. The communication rates of the distributed memory machines are high for all message sizes, as would be expected since these platforms are optimized for message-passing. However, it should also be noted that the communication rates on the shared memory machines are also reasonably high for medium and large-sized messages. Note the slower communication rate across a network of IBM RS6000 workstations. Also, note the very high communication rate between processes on a single IBM RS6000 workstation.

As described earlier, APPL uses a shared memory implementation of message-passing on shared memory machines. Although a socket-based implementation is simpler, and has been implemented in more popular message-passing libraries such as PVM (ref. 17), it has proven to be considerably slower than a shared memory implementation as illustrated by the communication rates calculated using a socket-based implementation of APPL in figure 11. This poorer performance is also evidenced in real applications. When the ISTAGE code, an inviscid average passage turbomachinery code, was executed on an SGI 4D/440 using both implementations of message-passing, the socket-based implementation proved to be 1.34 times slower than the shared memory-based version.

Application Performance

In porting an application across parallel computer platforms, a variety of architectures are encountered. Past experience has indicated that significant tuning of an application to a particular architecture is required in order to obtain the best performance. However, this process renders an application highly machine specific. Because we were interested in comparing the performance of an application on one machine versus another when using a portable message-passing library, the ISTAGE code and the MSTAGE code were run across several platforms. These large-scale codes are typical of those developed and used at NASA Lewis.

As mentioned above, ISTAGE is an inviscid average passage turbomachinery code, configured for a single blade row. A 120 x 11 x 11 grid was used for this timing study, and a medium grain partitioning was used for the domain decomposition. Figure 12 illustrates the performance of the portable, parallel ISTAGE code on three distributed memory machines, and figure 13 illustrates the performance on two shared memory machines and a network of workstations. Note that the ISTAGE code did not need to be modified when the code was moved across these diverse platforms.

Several observations can be made about the performance of ISTAGE on the various target machines. The ISTAGE code performs well across all of the distributed memory machines on which it was run, until load-balancing becomes an issue (ref. 10). A network of workstations can be considered a distributed memory platform; however, the performance of ISTAGE is poor on this platform due to the slower communication mechanism between processors. ISTAGE performs poorly on the shared memory machines as well, due to bus contention as processes communicate. Although the communication rates on these platforms are reasonable, neither the shared memory machines nor a network of workstations have parallel communication links between processors, and this absence adversely affects the performance of applications which require a significant amount of communication in parallel.

The MSTAGE code is a viscous average passage turbomachinery code, configured for multiple blade rows. The simulation used for this timing study featured four blade rows. A 218 x 31 x 31 grid was used, and a coarse grain partitioning was used for the domain decomposition. This is the "blade row parallel" version, which calculates each blade row on a separate processor. Because of the memory requirements per processor of this version of the code, there was a limit to the number of platforms to which the parallel MSTAGE code could be ported. In particular, the code was restricted from running on commercial distributed memory machines such as the Intel iPSC/860. Figure 14 illustrates the performance of the portable, parallel MSTAGE code on three parallel machines, including a distributed memory machine, a shared memory machine, and a network of workstations. Speedups in the range of 3.73 to 4 were achieved on four processors of these three parallel machines. Again, the MSTAGE code did not need to be modified when the code was moved across these diverse platforms. Thus the portable, parallel MSTAGE code performs well across all three diverse target machines.

The "blade row parallel" version of MSTAGE is limited to using only four processors for a four blade row simulation. Two techniques were used to extend the MSTAGE code to enable using more processors, and thereby improving performance. On the Hypercluster, a modified MSTAGE code uses 2 processors per blade row by splitting the domain along the K (tangential) dimension. Figure 15 illustrates the improved performance across eight processors. On the SGI 4D/480 machine, the Power Fortran Accelerator (ref. 11) was used to allow 2 processors per blade row to run DO loop iterations in parallel, in conjunction with the APPL parallelism. These improved results across eight processors are also illustrated in figure 15.

These timing studies indicate that one must carefully match a parallel application to a hardware platform. Coarse grain partitioned applications appear to demonstrate good performance on all available platforms. Medium and fine grain partitioned applications perform well on various distributed memory machines, but may incur too much communication overhead to demonstrate performance improvements on shared memory machines or on a network of workstations. Also, factors such as memory availability may restrict

an application from running on a particular platform. It has been concluded, however, that using a portable library simplifies the task of matching a parallel application to a hardware platform.

In addition, although the performance of fine and medium grain partitioned applications may be poor on both shared memory machines and a network of workstations, it was concluded that these platforms do provide an excellent code development environment. On several occasions, the use of a single workstation or a network of workstations with a stable operating environment helped to eliminate programming errors before the application code was moved to a larger parallel platform. Also, using a single processor workstation to simulate a parallel environment helps to encounter errors one at a time, rather than in parallel, making codes easier to debug step by step. Although this technique will not eliminate all programming errors, it can reduce the number of errors that will be encountered in the parallel environment. Finally, in the future, when thinking in terms of a network of distributed, heterogeneous machines, the use of a shared memory machine or a high-end workstation in the computational environment might actually be required. APPL provides a consistent programming interface across all of these machines.

CONCLUDING REMARKS

The Application Portable Parallel Library has proven to be a portable, user-friendly message-passing library. APPL has been targeted to a large number of MIMD machines (both shared and distributed memory), as well as networks of homogeneous workstations. Targets include the Intel iPSC/860, the Intel Delta, the Silicon Graphics 4D series, the IBM RS6000 series, the Sun Sparcstation series, the Alliant FX/80, and the Hypercluster (a NASA Lewis multi-architecture test bed). Applications programmed in Fortran or C using APPL have demonstrated low overhead when compared to results using the native programming environment. Also, APPL's implementation of message-passing using shared memory on shared memory machines has achieved better performance than socket-based message-passing implementations.

Timing studies comparing the performance of two large scale turbomachinery codes across a variety of parallel computer platforms have indicated that one must carefully match a parallel application to a hardware platform. Using a portable library simplifies this task. Also, shared memory machines and networks of workstations have proven to be ideal parallel code development environments.

APPL is portable across a variety of architectures, and may continue to be enriched with additional target implementations, as well as additional primitives as needed. Other future work being considered includes improving the ability to monitor performance, and extending APPL to run across a network of heterogeneous machines.

REFERENCES

1. Geist, G. A., Heath, M. T., Peyton, B. W., Worley, P. H., "PICL: A Portable Instrumented Communication Library", Oak Ridge National Laboratory, ORNL/TM-11130, March, 1990.

2. Boyle, J., et. al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., New York, NY, 1987.

3. Harrison, R. J., "Portable Tools and Applications for Parallel Computers," Int. J. Quant. Chem., 40 (1991), 847-863.

4. Butler, R., Lusk, E., "User's Guide for the P4 Parallel Programming System," Argonne National Laboratory, ANL-92/17, 1992.

5. Walker, D. W., "Standards for Message-Passing in a Distributed Memory Environment," Oak Ridge National Laboratory, ORNL/TM-12147, August, 1992.

6. Intel Corporation, *iPSC/2 and iPSC/860 C Language Reference Manual*, June 1990.

7. Rochkind, M. J., *Advanced UNIX Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

8. Akay, H. U., et. al., "A Database Management System for Parallel Processing of CFD Algorithms," *Proceedings of the Conference on Computational Fluid Dynamics*, Elsevier Science Publishers, The Netherlands, 1992.

9. Intel, Inc., *iPSC/860 Parallel Performance Analysis Tools Manual*, 1991.

10. Blech, R. A., et. al., "Turbomachinery CFD on Parallel Computers," NASA TM-105932, December 1992.

11. Silicon Graphics, Inc., *POWER Fortran Accelerator User's Guide,* Doc. #007-0715-030, January 1992.

12. Silicon Graphics, Inc., *IRIX User's Reference Manual*, Vol. 1, Section 1, Doc. #007-0606-050, April 1990.

13. Gerald, C. F., *Applied Numerical Analysis*, 2nd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts, May 1980, pp. 340-356.

14. Townsend, S. E., Blech, R. A., and Cole, G. L., "A Multiarchitecture Parallel-Processing Development Environment," NASA TM-106180, 1993.

15. Adamczyk, J. J., et. al., "Simulation of Three Dimensional Viscous Flow within a Multistage Turbine," Transactions of the ASME, Vol. 112, July 1990.

16. Chima, R. V., "Development of an Explicit Multigrid Algorithm for Quasi-Three-Dimensional Viscous Flows in Turbomachinery," AIAA Paper No. 86-0032; NASA TM-87128, 1986.

17. Sunderam, "PVM: A Framework for Parallel Distributed Computing," Concurrency: Practice & Experience, Vol. 2, No. 4, December 1990.

void **pbegin**()

- Initializes the environment. The first executable statement in each user program.

void **pend**()

- Cleans up the environment. The last executable statement in each user program.

int **myid**()

- Returns the process id of the calling process.

int **nprocs**()

- Returns the number of user processes.

void **ssend** (int msgtype, char *msg, int length, int process_id)

- A fully blocking send operation, which must be used in conjunction with **srecv**. A buffer pointed to by *msg*, of size *length* and message type *msgtype*, is sent to *process_id*. Execution is suspended until *process_id* acknowledges the receipt of the message.

void **srecv** (int type_cond, char *msg, int length)

- A blocking receive operation, which must be used in conjunction with **ssend**. The incoming message is stored in *msg*, and is of size <= *length*, and has a message type of *type_cond*. If *type_cond*= -1, the next available message is selected. On completion of the receive, an acknowledgement is returned to the sender.

void **asend** (int msgtype, char *msg, int length, int process_id)

- A partially blocking send operation which must be used in conjunction with **arecv**. A buffer pointed to by *msg*, of size *length* and message type *msgtype*, is sent to *process_id*. Execution continues once the send has been initiated.

void **arecv** (int type_cond, char *msg, int length)

- A blocking receive operation, which must be used in conjunction with **asend**. The incoming message is stored in *msg*, and is of size <= *length*, and has a message type of *type_cond*. If *type_cond*= -1, the next available message is selected. No acknowledgement is returned to the sender.

int **probe** (int type_cond)

- A non-blocking probe operation which determines if a message of type= *type_cond* is available to be received. If *type_cond*= -1, the next available message is selected. Returns 1 if such a message is found, and 0 if not.

double **what_time**()

- Returns the value of a counter which reflects the relative time in milliseconds.

int **info_pid**()

- Returns the process id of the last message received or probed.

int **info_len**()

- Returns the length of the last message received or probed.

int **info_type**()

- Returns the message type of the last message received or probed.

Table 1: APPL Low-Level Primitives

void **bcast** (int msgtype, char *msg, int length, int root)

- Broadcast *msg* of size *length* and type *msgtype*. If *root*= **myid**(), then the calling process is the broadcasting process. If *root* != **myid**(), then the calling process is a receiving process.

void **dgsum** (double *x, long n, double *work)

- Computes the global sum of x across all processes. X is a vector of length n. Each element of the resulting vector is the sum of the corresponding elements of the input vectors of each process. The result is returned in x on all processes. Since this is a global operation, all processes must execute this operation before the computation can continue. Also, **igsum** exists for integers, and **rgsum** for real (single precision) numbers.

void **dgprod** (double *x, long n, double *work)

- Computes the global product of x across all processes. X is a vector of length n. Each element of the resulting vector is the product of the corresponding elements of the input vectors of each process. The result is returned in x on all processes. Since this is a global operation, all processes must execute this operation before the computation can continue. Also, **igprod** exists for integers, and **rgprod** for real (single precision) numbers.

void **dgmax** (double *x, long n, double *work)

- Computes the global maximum of x across all processes. X is a vector of length n. Each element of the resulting vector is the maximum of the corresponding elements of the input vectors of each process. The result is returned in x on all processes. Since this is a global operation, all processes must execute this operation before the computation can continue. Also, **igmax** exists for integers, and **rgmax** for real (single precision) numbers.

void **dgmin** (double *x, long n, double *work)

- Computes the global minimum of x across all processes. X is a vector of length n. Each element of the resulting vector is the minimum of the corresponding elements of the input vectors of each process. The result is returned in x on all processes. Since this is a global operation, all processes must execute this operation before the computation can continue. Also, **igmin** exists for integers, and **rgmin** for real (single precision) numbers.

Table 2: APPL High-Level Primitives

NASA Lewis

- ISTAGE inviscid average passage turbomachinery code
- MSTAGE viscous average passage turbomachinery code
- MHOST general finite element structural analysis program
- PARC3D CFD code

Indiana University-Purdue University at Indianapolis

- Grid-oriented database for parallel processing
- ADPAC turbomachinery code

University Space Research Association (NASA Goddard)

- Gravitational n-body simulations of interacting and merging galaxies

NASA Langley

- VGRIDSG unstructured surface grid generation program

Table 3: Applications developed using APPL

Machine	SMALL (8 B) Native	APPL	Overhead	MEDIUM (2 KB) Native	APPL	Overhead	LARGE (16 KB) Native	APPL	Overhead
Intel iPSC	0.126	0.147	16.7%	1.564	1.580	1.0%	9.743	9.763	0.2%
Intel Delta	0.110	0.136	23.6%	0.679	0.703	3.5%	3.704	3.819	3.1%
Hypercluster	0.215	0.231	7.4%	1.994	2.010	0.8%	12.123	12.212	0.7%

a. Ring test: 16 processors, 100 iterations, wall clock time in seconds

Machine	Native	APPL	Overhead
Intel iPSC	6.002	6.036	0.6%
Intel Delta	5.420	5.609	3.5%
Hypercluster	6.500	6.619	1.8%

b. 2-D Euler code: 16 processors, 100 iterations, 65 x 65 grid, wall clock time in seconds

Table 4: Overhead incurred using APPL rather than each machine's native message-passing library

Intel iPSC/860:

```
fsang    babbage    laplace                -2   4
```

Alliant FX/80:

```
fsang    alliant1   /usr/lerc/fsang/work   4    laplace laplace laplace laplace
```

Network of IBM RS6000s:

```
fsang    lace20     /home/fsang/work       1    laplace
fsang    lace21     /home/fsang/demo       1    laplace
guest    hercules   /u/guest/work          1    laplace
fsang    lace28     /home/fsang/work       1    laplace
```

Figure 1: Sample process definition files

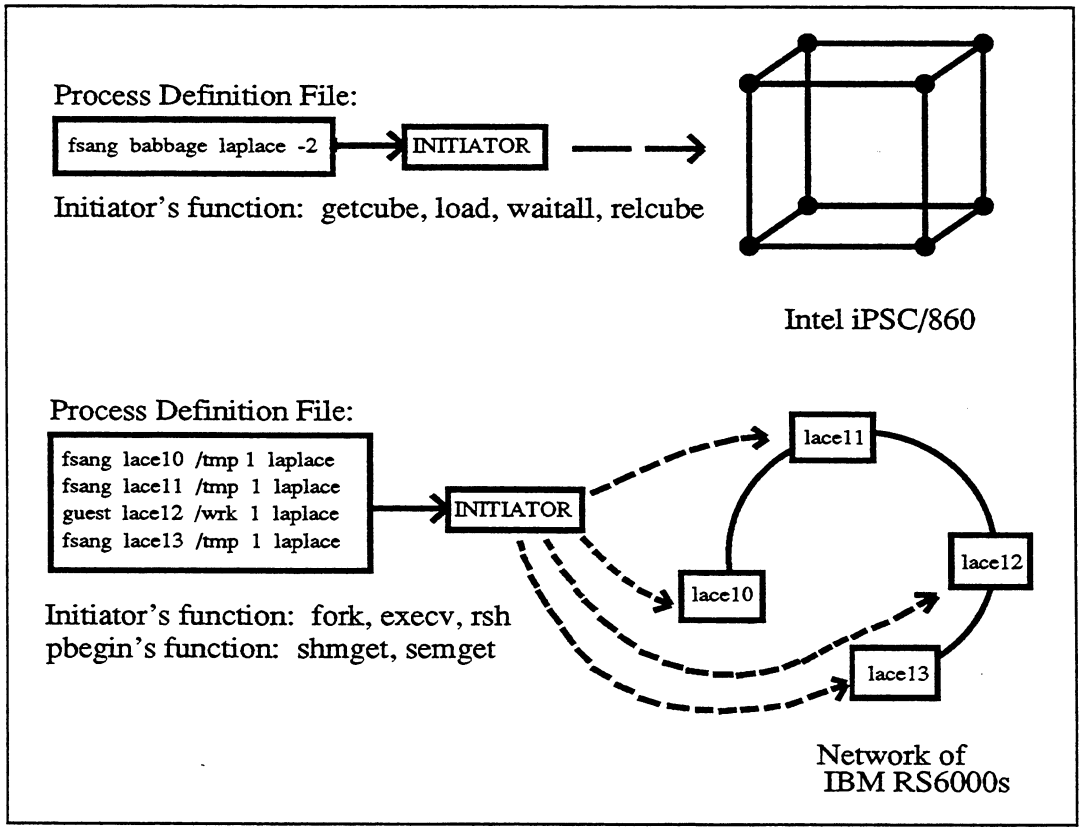

Figure 2: Function of the Initiator Process

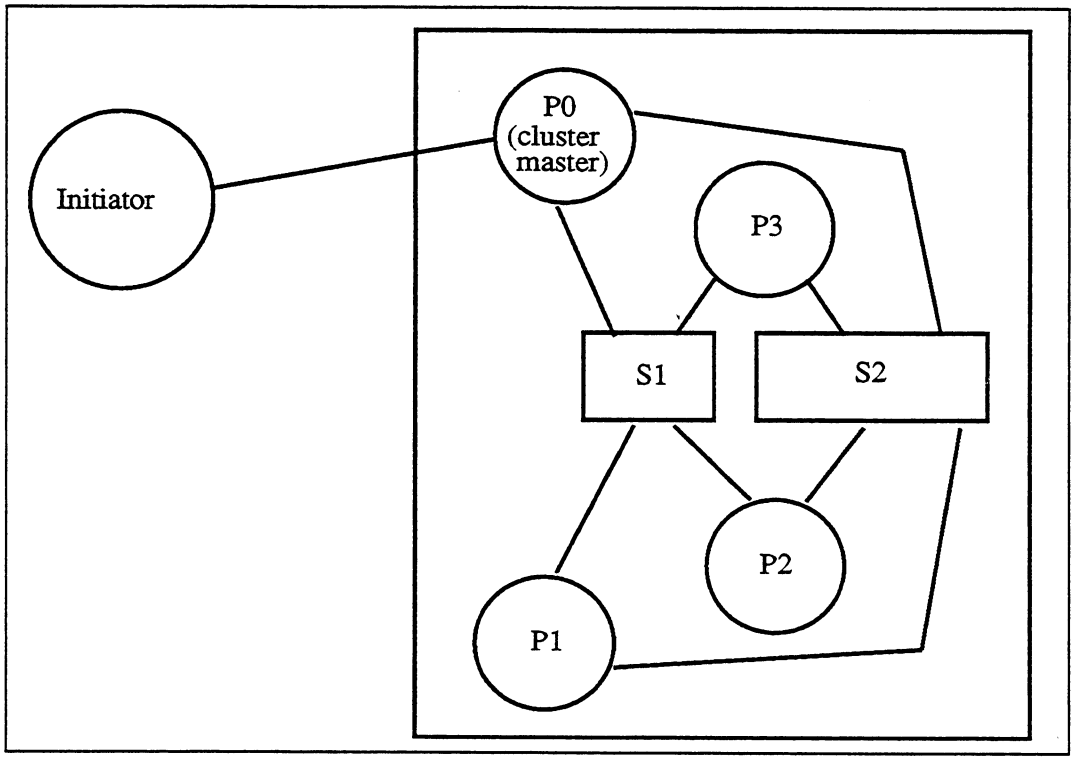

Figure 3: Process cluster

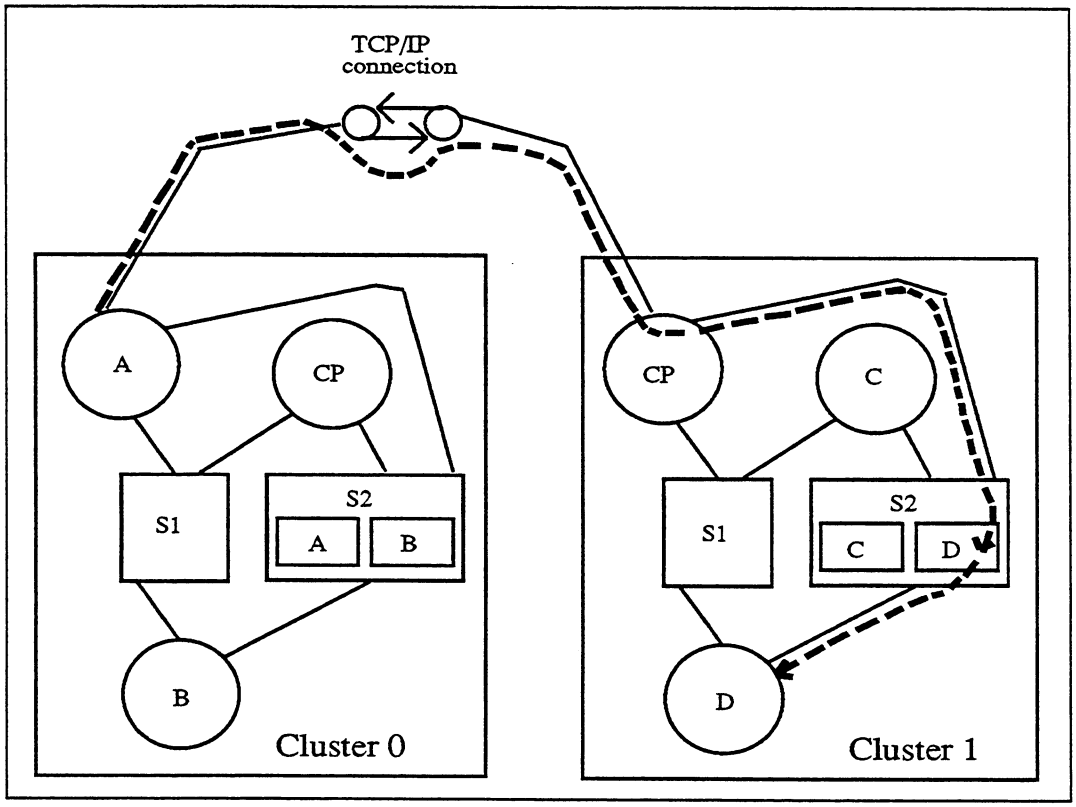

Figure 4: Message transmission between clusters

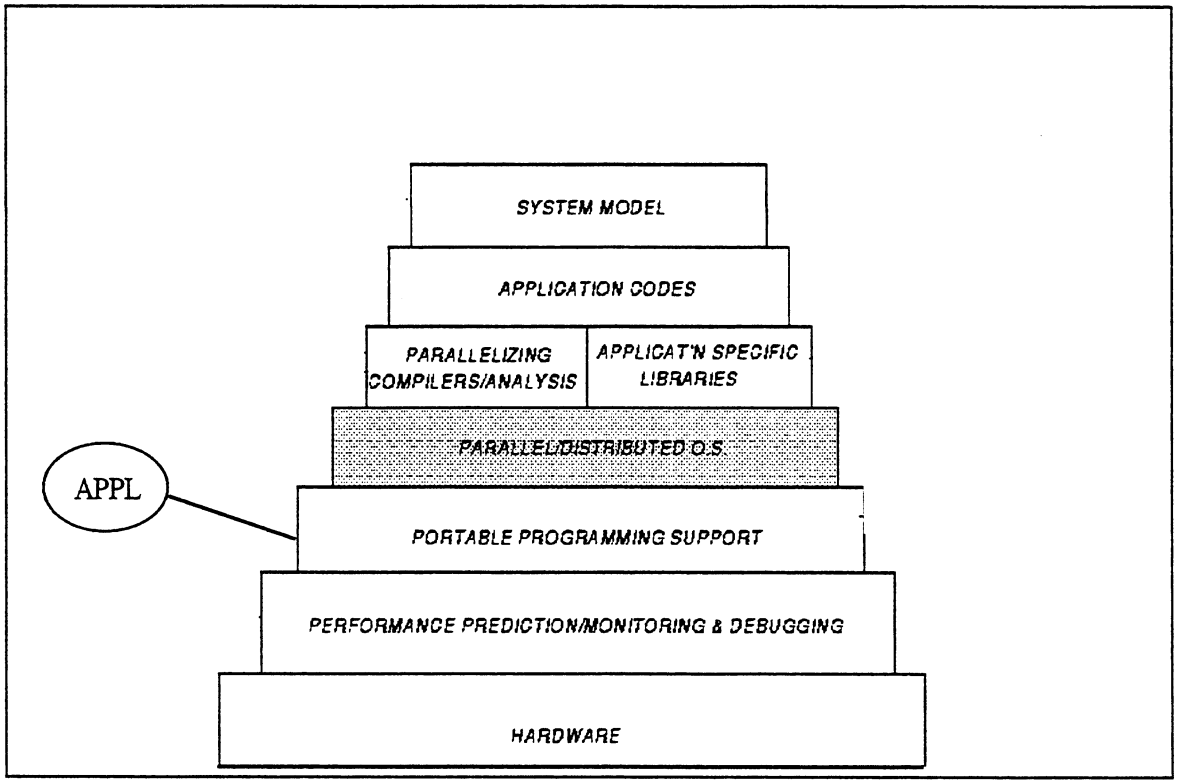

Figure 5: Software Infrastructure

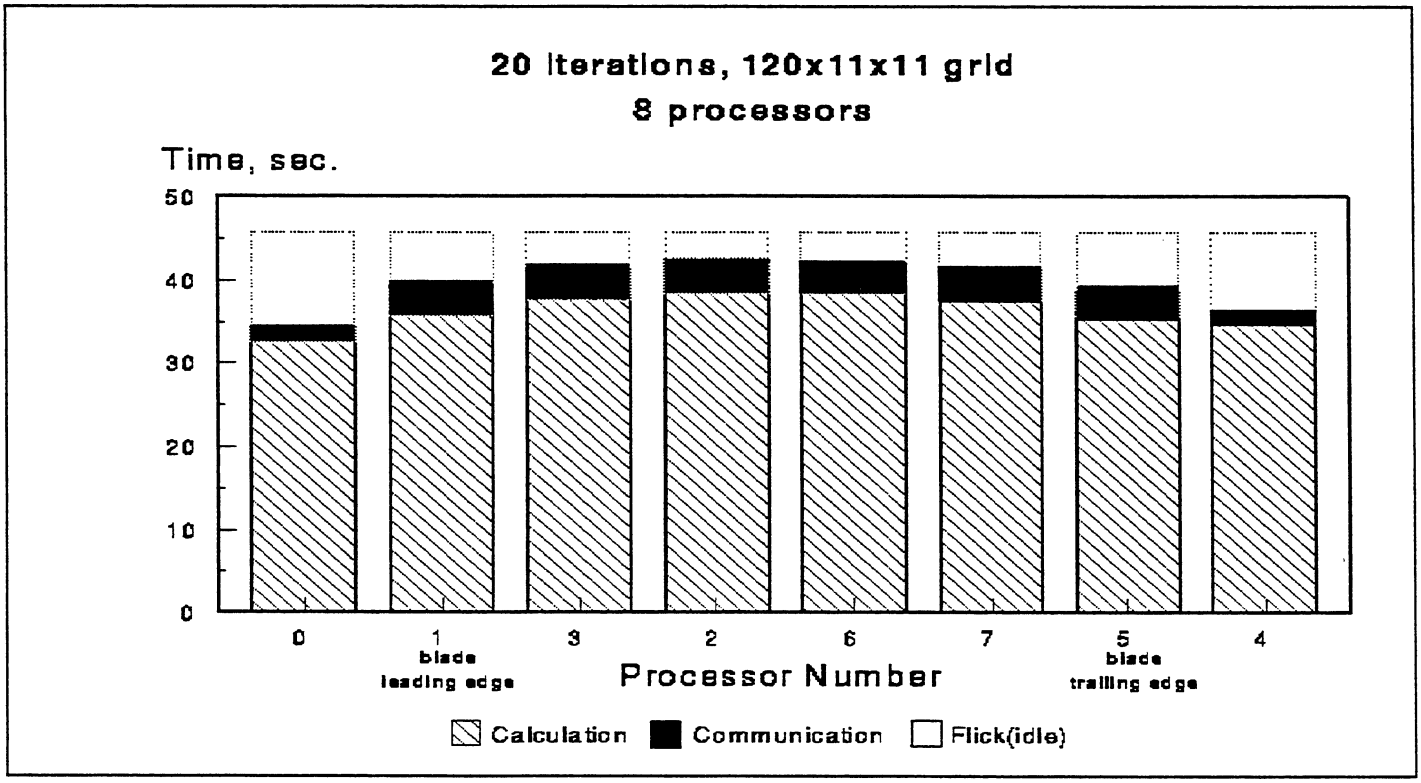

Figure 6: Performance statistics for parallel ISTAGE code, generated by Intel's Performance Analysis Tool

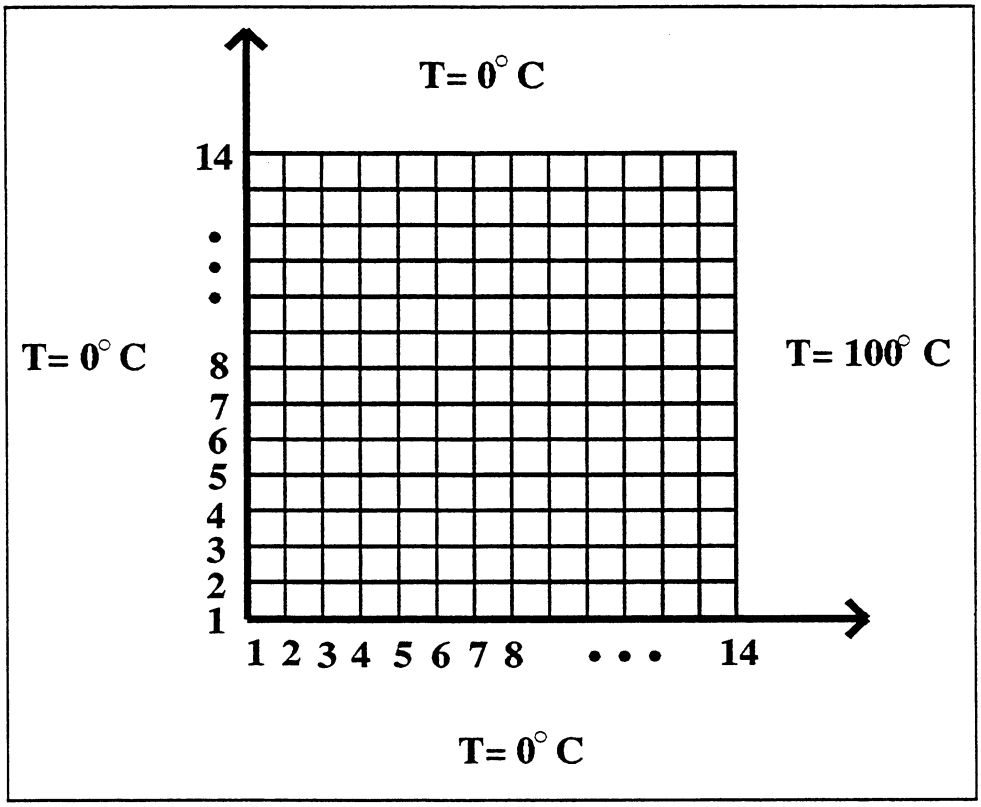

Figure 7: Discretized example problem

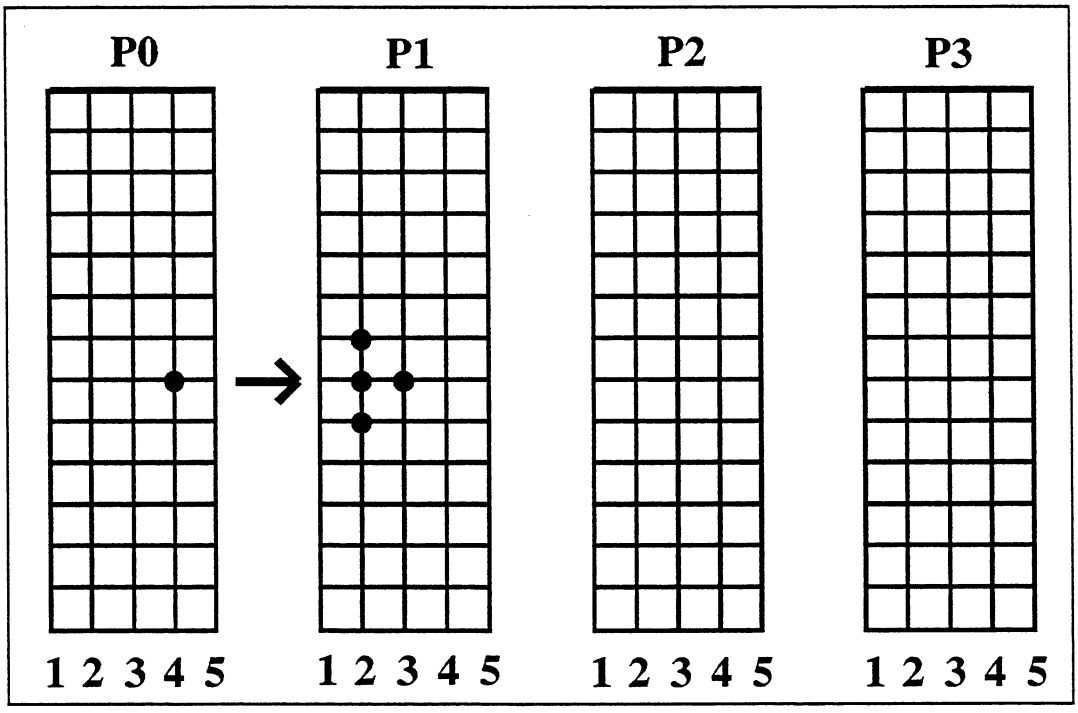

Figure 8: Dividing the problem using four processes

```
C   The following parameters were read or calculated by the calling process, and were passed to the
C   subroutines below via a common block (include 'laplace.h').

C       rows                    Total number of grid rows
C       cols                    Total number of grid columns
C       my_col                  Number of grid columns for my process.  Each process has its share of the grid,
C                               plus two boundary columns to store incoming columns from its neighbors:
C                                       my_col= ((cols-2) / nprocs()) + 2
C                               The value of my_col should be a whole number in order for this example to work correctly.
C       north, south, east, west Boundary temperatures
C       interior                Interior temperature
C       max_iter                Maximum iterations

      subroutine laplace(temp_a, temp_b)
      include 'laplace.h'
      double precision temp_a(rows, my_col), temp_b(rows, my_col)

      call pbegin

C     Determine neighboring processes
      right= mod((myid() + 1), nprocs())
      left= myid() - 1
      if (right .eq. 0) right= -1

      call InitializeGridTemperatureValues ()

C     Main iteration loop:  compute two iterations to swap from temp_a to temp_b and back.
      do 1 iterations= 1, (max_iter/2)
           call do_iteration (temp_a, temp_b)
           call do_iteration (temp_b, temp_a)
1     continue

      call PrintResults ()

      call pend
      end

      subroutine do_iteration (old, new)
      include 'laplace.h'
      double precision old(rows, my_col), new(rows, my_col)

      LCOL= 10
      RCOL= 20
      num_bytes= (rows-2)*8

C     Transmit data to neighbors
      if (right .ne. -1) call asend (RCOL, old(2,my_col-1), num_bytes, right)
      if (left .ne. -1) call asend (LCOL, old(2,2), num_bytes, left)

C     Receive data from neighbors
      if (right .ne. -1) call arecv (LCOL, old(2, my_col), num_bytes)
      if (left .ne. -1) call arecv (RCOL, old(2, 1), num_bytes)

C     Perform calculation on my section of grid
      do 2 j= 2, my_col-1
           do 3 i= 2, rows-1
                new(i, j)= (old(i+1,j) + old(i-1, j) + old(i, j+1) + old(i, j-1)) / 4.0
3          continue
2     continue
      end
```

Figure 9: Pseudo-code segment

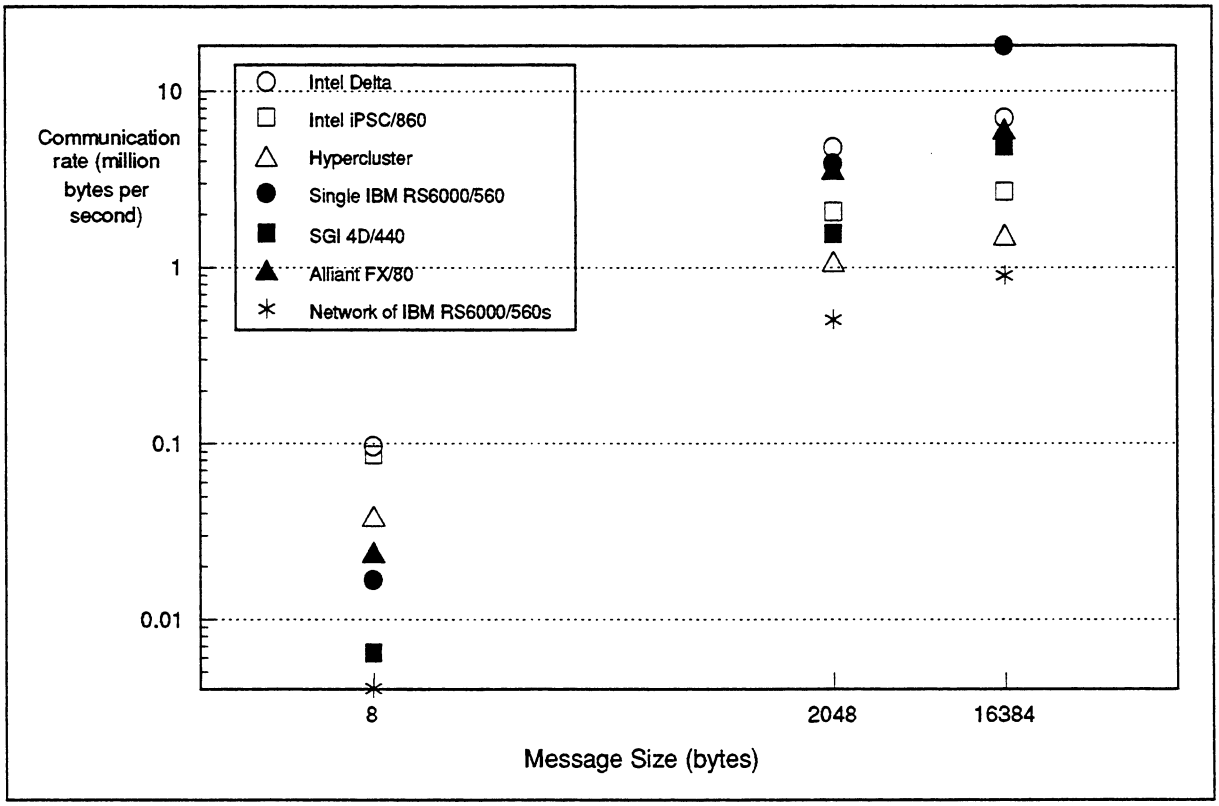

Figure 10: Communication rates using APPL

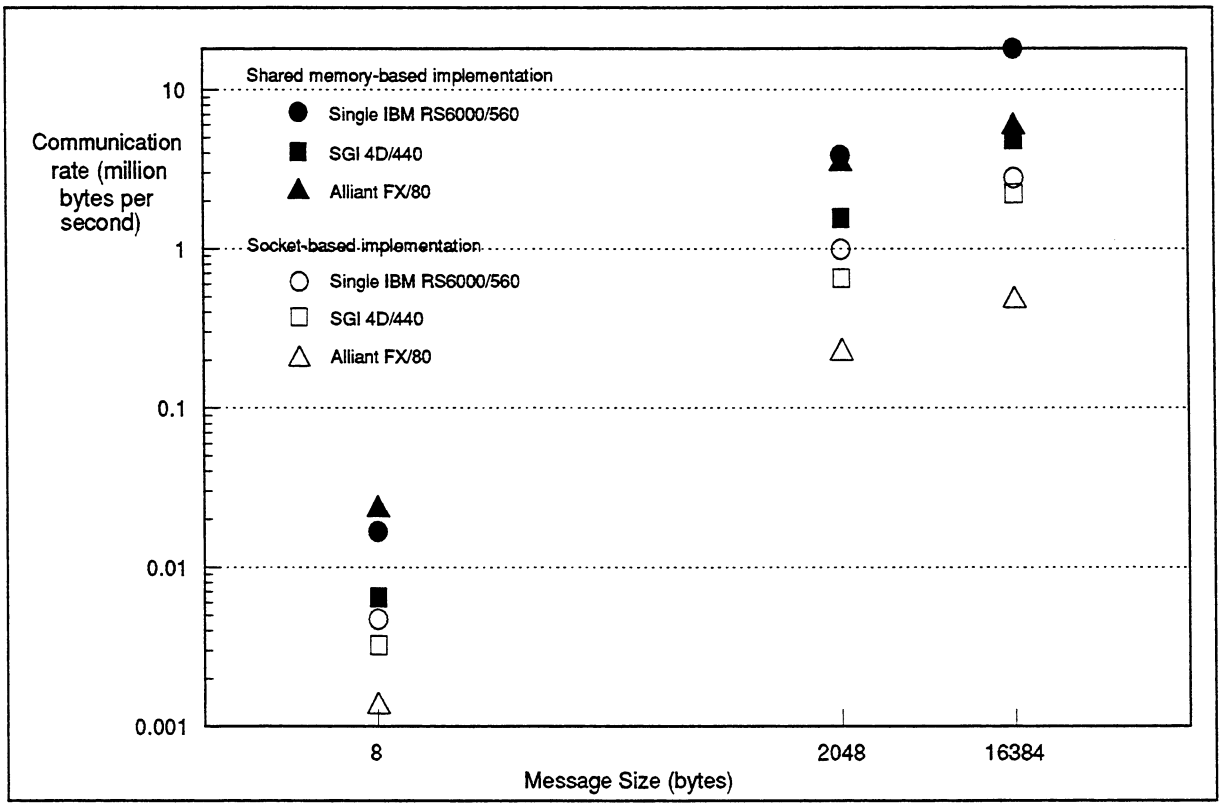

Figure 11: Comparison of communication rates using both a shared memory and a socket-based implementation of message-passing

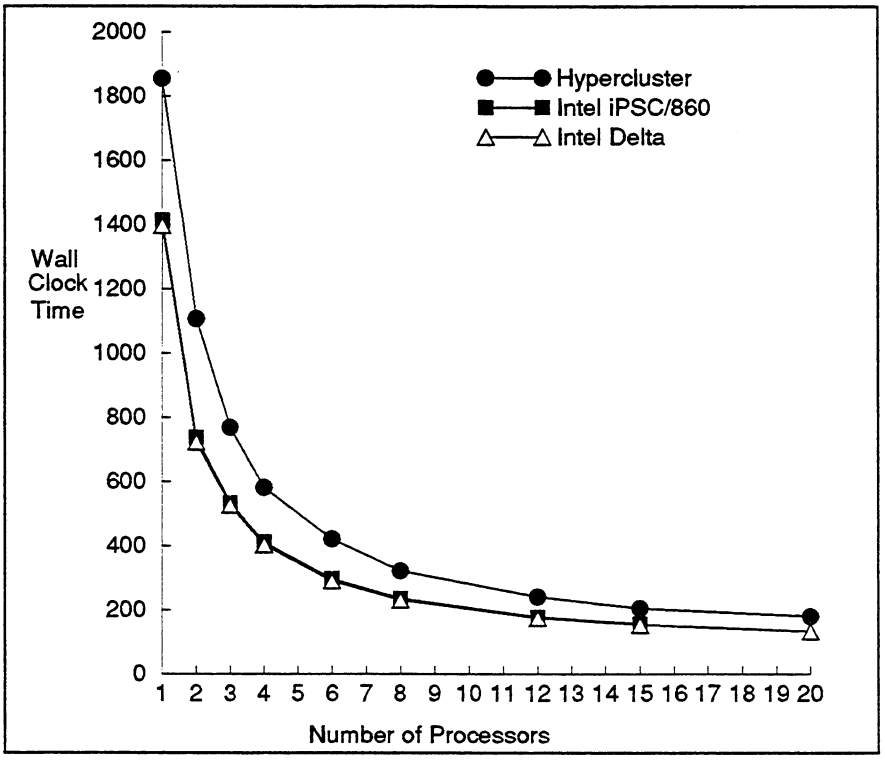

Figure 12: ISTAGE performance results on distributed memory machines

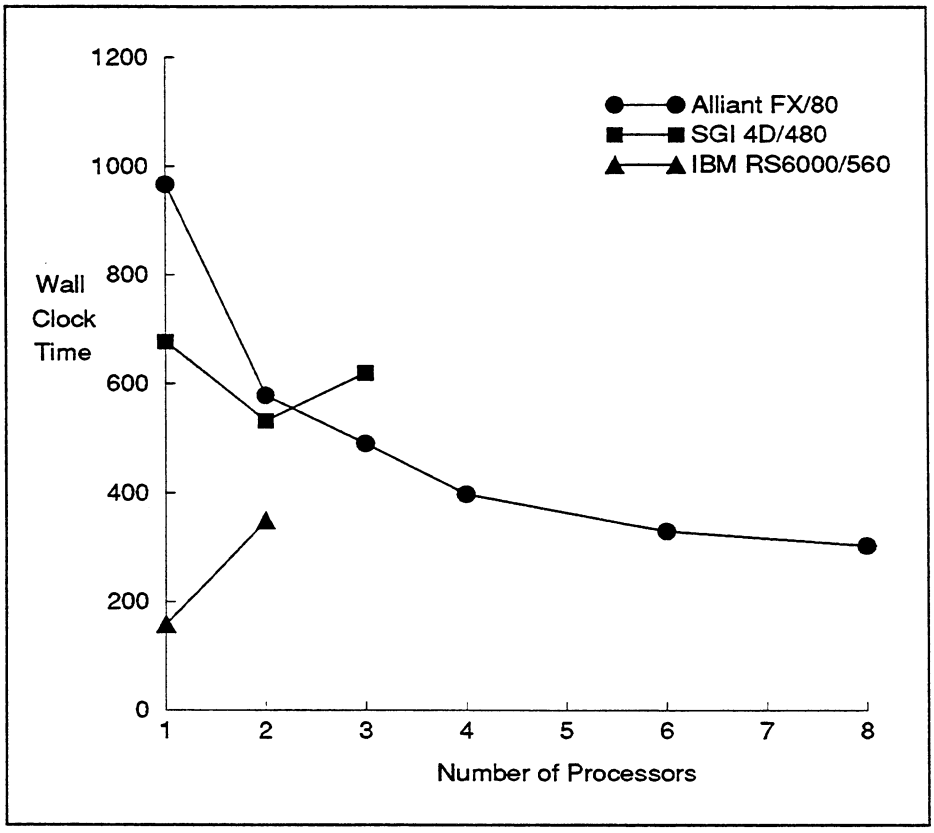

Figure 13: ISTAGE performance results on shared memory machines and a network of workstations

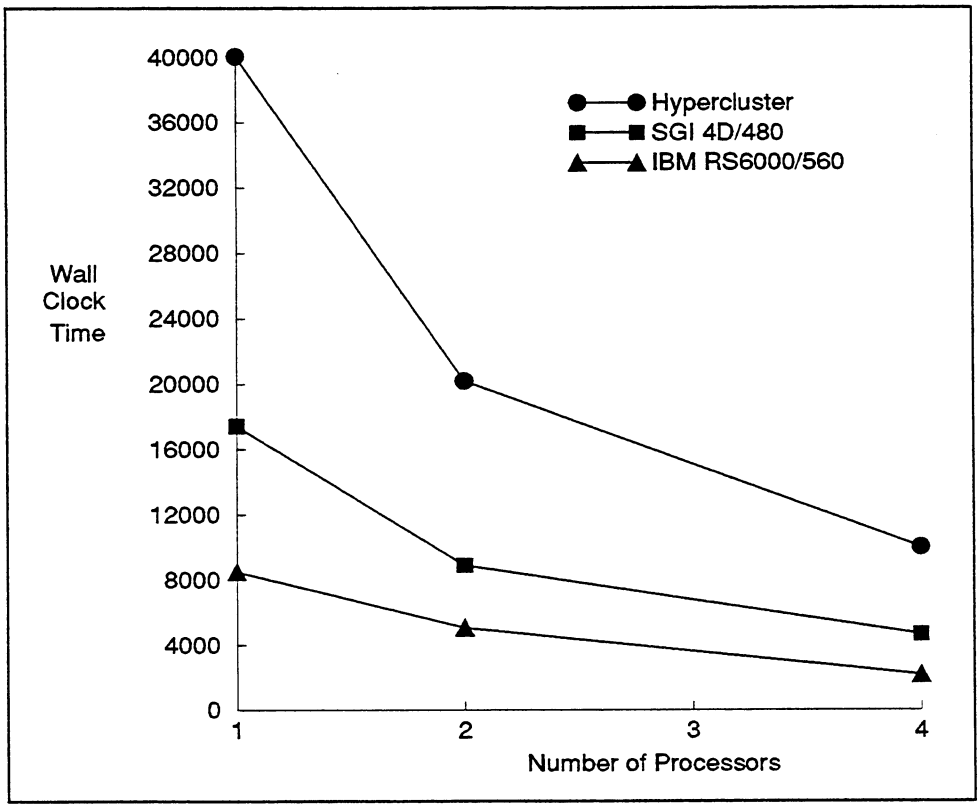

Figure 14: MSTAGE performance results

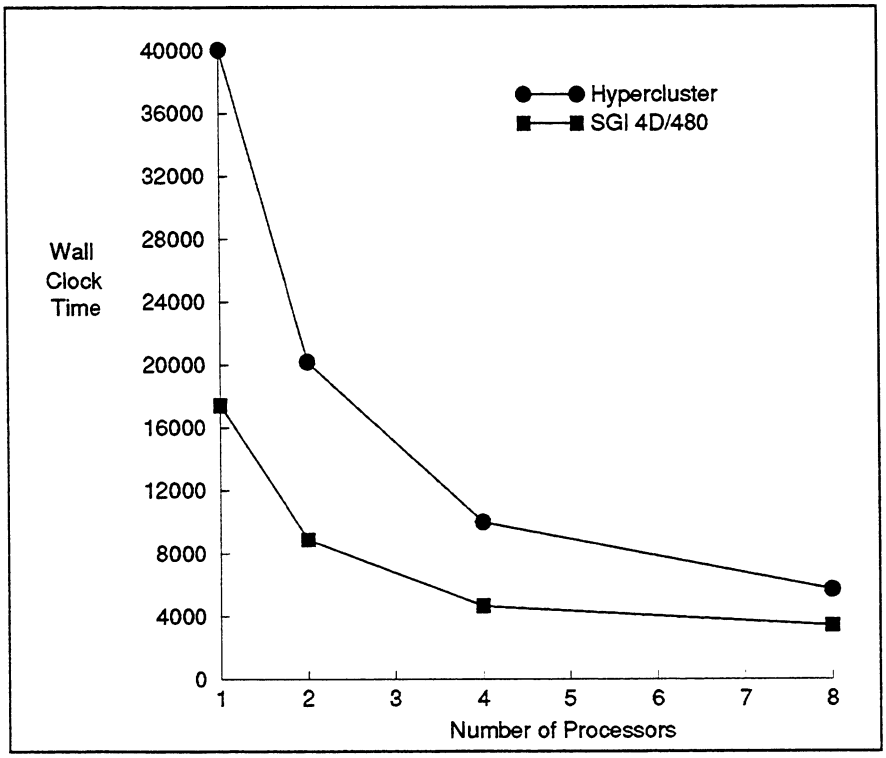

Figure 15: Extended MSTAGE performance results

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1993	Technical Memorandum

4. TITLE AND SUBTITLE

Portable Programming on Parallel/Networked Computers Using the Application Portable Parallel Library (APPL)

5. FUNDING NUMBERS

WU-505-62-52

6. AUTHOR(S)

Angela Quealy, Gary L. Cole, and Richard A. Blech

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E-7960

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM-106238

11. SUPPLEMENTARY NOTES

Angela Quealy, Sverdrup Technology, Inc., Lewis Research Center Group, 2001 Aerospace Parkway, Brook Park, Ohio 44142; and Gary L. Cole and Richard A. Blech, NASA Lewis Research Center. Responsible person, Angela Quealy, (216) 826-6642.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

The Application Portable Parallel Library (APPL) is a subroutine-based library of communication primitives that is callable from applications written in FORTRAN or C. APPL provides a consistent programmer interface to a variety of distributed and shared-memory multiprocessor MIMD machines. The objective of APPL is to minimize the effort required to move parallel applications from one machine to another, or to a network of homogeneous machines. APPL encompasses many of the message-passing primitives that are currently available on commercial multiprocessor systems. This paper describes APPL (version 2.3.1) and its usage, reports the status of the APPL project, and indicates possible directions for the future. Several applications using APPL are discussed, as well as performance and overhead results.

14. SUBJECT TERMS

Parallel processing (computers); Message-passing; Portability; Interprocessor communication; Distributed processing; Portable CFD applications

15. NUMBER OF PAGES

25

16. PRICE CODE

A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	